FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors (Version 1.2)

Rudolf Berrendorf, Heinz Ziegler

FZJ-ZAM-IB-9816

Oktober 1998

(letzte Änderung: 19.08.99)

PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors (Version 1.2)

Rudolf Berrendorf, Heinz Ziegler
Central Institute for Applied Mathematics
Research Centre Juelich GmbH
D-52425 Juelich, Germany
r.berrendorf@fz-juelich.de

Abstract

A performance counter is that part of a microprocessor that measures and gathers performance-relevant events on the microprocessor. The number and type of available events differ significantly between existing microprocessors, because there is no commonly accepted specification, and because each manufacturer has different priorities on analyzing the performance of architectures and programs. Looking at the supported events on the different microprocessors, it can be observed that the functionality of these events differs from the requirements of an expert application programmer or a performance tool writer.

PCL, the Performance Counter Library, establishes a common platform for performance measurements on a wide range of computer systems. With a common interface on all systems and a set of application-oriented events defined, the application programmer is able to do program optimization in a portable way and the performance tool writer is able to rely on a common interface on different systems.

PCL has functions to query the functionality, to start and to stop counters, and to read the values of counters. Performance counter values are returned as 64 bit integers on all systems. PCL supports nested calls to PCL functions thus allowing hierarchical performance measurements. Counting may be done either in system or in user mode. All interface functions are callable in C, C++, Fortran, and Java.

Contents

Chapter 1

Introduction

This report describes performance counters on 5 microprocessor families and introduces a common interface to access these counters. With performance counters, performance critical events can be counted. This includes all aspects concerning the memory hierarchy (loads/stores, misses/hits, different cache levels, etc.), functional units or pipelines (operation counts, stalls, issues), duration of requests, etc.

As will be shown, the number of, type of, and access to events differs significantly between the processors and the type of supported events might be not very helpful to the application programmer or tool builder who might have different demands of countable events.

To overcome this lack of common platform, we developed PCL, the Performance Counter Library. We first defined a set of events useful to the application programmer and tool builder, and second, established a set of access functions to control and access the performance counters on different platforms. PCL is implemented on many of todays machines ranging from a PC running Linux to a SGI/CRAY T3E with hundreds of Gigaflops and it is callable from application programs as well as from tools.

The Performance Counter Library PCL is available at

http://www.fz-juelich.de/zam/PCL/.

Chapter 2

Requirements of Application Programmers

People from different areas of computer science and electrical engineering may see different events as most useful for their optimization purposes. Most of the events described so far in the description of the microprocessors are likely most useful to the computer architect, hardware engineer, or low-level device driver writer.

Application programmers optimizing their programs or performance tool writers wish to get performance relevant information related to their programs rather than counting signal switches on certain pins of a chip module. Therefore, those parts of the microprocessor which have appropriate counterparts in a program are most likely to be used by the application programmer to optimize programs. The memory hierarchy in a computer system corresponds directly to program variables and the functional units execute the operations specified in a program. Therefore, we concentrate on those aspects of a computer system.

Our impression is, that taking the union of all available events of all microprocessors is not the right way to define an application interface for an application programmer or tool writer. Our approach is to define a set of events *relevant* to the user. If microprocessor architecture or programming methodology precedes in a different direction (we don't see that for the near future!), the set of events might then be extended or changed.

Although hardware counters give numbers for a processor, performance numbers should be related to a process (representing the program). Therefore, either the executing process should be bound to a processor, or migrating a process to another processor should be transparent to the process (related to performance counting). Using the second approach needs support of the operating system.

We have categorized the useful events into categories as shown in the following sections.

2.1 Memory Hierarchy

Currently, most computer systems support four levels in the memory hierarchy: registers, 1st level cache, 2nd level cache, main memory. Registers are directly controlled by a compiler, so for example, the information how many registers keep live values could be better managed by a compiler. Although main memory statistics could be quite useful in performance analysis (e.g. bank conflicts), performance counters in microprocessors mostly see the main memory as a black box. Therefore, we concentrate on 1st and 2nd level caches.

Accesses to caches can be distinguished by read or write accesses, instruction loads and instruction stores (fetches from a higher level in the hierarchy), or data load/stores. An important performance aspect is the hit and miss rate, which can be calculated from the total number of accesses and either the number of misses or hits. Most microprocessors use (small) translation look-aside buffers (TLB) to speed up the translation of virtual to physical addresses. As misses in the TLB are time consuming, this number (and its relation to the number of hits or the total number of address lookups) is a relevant number for performance optimization.

We distinguish between instruction and data caches on each level. For unified caches (i.e. instruction and data are buffered in the same cache), it is often possible to distinguish instruction and data loads. Therefore on those caches, *PCL_LxICACHE_xxx* and *PCL_LxDCACHE_xxx* refer to events concerning instruction and data accesses, respectively.

The available events concerning memory hierarchy are given in table 2.1.

Due to the definition, the sum of cache reads and cache writes should be equal to cache read/writes and the the sum of cache hits and cache misses should be equal to cache read/writes, too. Additionally, if two

cache	
PCL_LxCACHE_READ	number of level-x cache reads
PCL_LxCACHE_WRITE	number of level-x cache writes
PCL_LxCACHE_READWRITE	number of level-x cache reads or writes
PCL_LxCACHE_HIT	number of level-x cache hits
PCL_LxCACHE_MISS	number of level-x cache misses
data cache	
PCL_LxDCACHE_READ	number of level-x data cache reads
PCL_LxDCACHE_WRITE	number of level-x data cache writes
PCL_LxDCACHE_READWRITE	number of level-x data cache reads or writes
PCL_LxDCACHE_HIT	number of level-x data cache hits
PCL_LxDCACHE_MISS	number of level-x data cache misses
instruction cache	
PCL_LxICACHE_READ	number of level-x instruction cache reads
PCL_LxICACHE_WRITE	number of level-x instruction cache writes
PCL_LxICACHE_READWRITE	number of level-x instruction cache reads or writes
PCL_LxICACHE_HIT	number of level-x instruction cache hits
PCL_LxICACHE_MISS	number of level-x instruction cache misses
TLB	
PCL_TLB_HIT	number of hits in TLB
PCL_TLB_MISS	number of misses in TLB
Instruction TLB	
PCL_ITLB_HIT	number of hits in instruction TLB
PCL_ITLB_MISS	number of misses in instruction TLB
Data TLB	
PCL_DTLB_HIT	number of hits in data TLB
PCL_DTLB_MISS	number of misses in data TLB

Table 2.1: Events concerning memory hierarchy (x=1 or 2 for 1st or 2nd level cache)

first level caches exist (instruction and data), the sum of instruction cache reads and data cache reads should be equal to cache reads (and so on).

2.2 Instructions

Instructions correspond to operations and flow control specified in a program. There are several categories of operations (e.g. integer, logical, floating point) which might be executed by different functional units in the microprocessor. Another aspect (in multiprocessor systems) is atomic operations (e.g. a primitive for a test-and-set-operations) which can be executed successful (the lock could be set) or unsuccessful (the lock could not be acquired as it was already set). We distinguish between the instruction categories as shown in table 2.2.

Additionally, we have included a cycle count which gives the number of cycles spent in this process or on behalf of the process/thread (when counting in user-and-system mode). For clarification, it should be noted that the cycle count should not be used to count the number of elapsed cycles as on multiprogramming systems other processes might be scheduled to the same processor. To count the number of elapsed cycles, an additional event can be used (*PCL_ELAPSED_CYCLES*).

On some systems, the number of issued instructions might be different to the number of completed instructions due to some error conditions. We have chosen completed instructions, as they correspond more closely to the operations the programmer specified in his program.

Getting the number of operations out of the number of instructions is difficult. For example, on some systems a floating-point add and a floating-point multiply can be initiated by a single add-and-multiply instruction. Therefore, 1 floating point instruction is counted but 2 floating point operations are executed. With PCL (and most of all hardware performance counter implementations) it is not possible to count the number of floating point *operations* and related number.

2.3 Status of Functional Units

Functional units might be stalled due to blocked resources, missing operands etc. Table 2.3 gives the events defined for stalls. Measuring such an event results (different to all other events) not in the number of stalls

PCL_CYCLES	spent cycles in process/thread (and eventually in system calls)
PCL_ELAPSED_CYCLES	elapsed cycles
PCL_INTEGER_INSTR	number of completed integer (or logical) instructions
PCL_FP_INSTR	number of completed floating point instructions
PCL_LOAD_INSTR	number of completed load instructions
PCL_STORE_INSTR	number of completed store instructions
PCL_LOADSTORE_INSTR	number of completed load or store instructions
PCL_INSTR	sum of all completed instructions
PCL_JUMP_SUCCESS	number of correctly predicted branches
PCL_JUMP_UNSUCCESS	number of mispredicted branches
PCL_JUMP	sum of all branches
PCL_ATOMIC_SUCCESS	number of successful atomic instructions
PCL_ATOMIC_UNSUCCESS	number of unsuccessful atomic instructions
PCL_ATOMIC	sum of all instructions concerning atomic operations

Table 2.2: Events concerning instruction categories

PCL_STALL_INTEGER	number of cycles the integer/logical unit is stalled
PCL_STALL_FP	number of cycles the floating point unit is stalled
PCL_STALL_JUMP	number of cycles the branch unit is stalled
PCL_STALL_LOAD	number of cycles the load unit is stalled
PCL_STALL_STORE	number of cycles the store unit is stalled (write buffer)
PCL_STALL	sum of all cycles a unit is stalled

Table 2.3: Events concerning functional unit stalls (numbers given in cycles)

but in the number of cycles all stalls of this event type have taken.

2.4 Rates and Ratios

Often, it is useful to get a ratio or rate rather than an absolute number. Good examples are cache miss rates or floating point operations per second. Table 2.4 gives the events defined for such rates and ratios.

Measuring these events will mostly be done by deriving the values from other performance numbers (see [1]). The definitions are as follows:

- PCL_MFLOPS : $\frac{PCL\_FP\_INSTR}{PCL\_CYCLES} \times MHzrate$
- PCL_IPC : $\frac{PCL\_INSTR}{PCL\_CYCLES}$
- PCL_L1DCACHE_MISSRATE : $\frac{PCL\_L1DCACHE\_MISS}{PCL\_LOADSTORE\_INSTR}$
- PCL_L2DCACHE_MISSRATE : $\frac{PCL\_L2DCACHE\_MISS}{PCL\_L1DCACHE\_MISS}$
- PCL_MEM_FP_RATIO : $\frac{PCL\_LOADSTORE\_INSTR}{PCL\_FP\_INSTR}$

PCL_MFLOPS	number of million floating point instructions per second
PCL_IPC	number of completed instructions per cycle
PCL_L1DCACHE_MISSRATE	miss rate of L1 data cache
PCL_L2DCACHE_MISSRATE	miss rate for L2 data cache
PCL_MEM_FP_RATIO	ratio of memory references to floating point operations

Table 2.4: Events concerning rates and rations (numbers are floating point values)

Chapter 3

PCL – The Performance Counter Library

The Performance Counter Library has a programming interface to access a set of performance counters with a defined set of countable events. In section 3.1, we specify which of the events defined in chapter 2 are available on what systems and in section 3.2 we define the programming interface.

3.1 Countable Events

In the following tables we compare the events defined in the last section in tables 2.1 to 2.3 with the available events on the microprocessors currently supported by PCL.

The tables are given in the following scheme. The first column gives the event family, followed by the precise event, one in each row. Each additional column contains entries for one microprocessor (family). The entry names correspond to the event names in the description of the microprocessors (see chapter A. Empty entries signal that such an event is not available on that microprocessor. Entries marked with a star are indirect events as a combination of several other events directly countable by a (hardware) performance counter. The combinations for these indirect events are discussed below. Counters used for indirect events can not be used at the same time to measure their own events.

Table 3.1 shows events relevant to the 1st level cache (instruction, data, instruction and data), table 3.2 shows events relevant to the 2nd level cache (instruction, data, instruction and data). If there is a unified cache for data and instructions (as it is on most systems), events defined for 2nd level instruction cache refer to cache references done by instruction fetches, and for the data cache accordingly. Table 3.3 shows events for the translation look-aside buffers (instruction, data, instruction and data). Table 3.4 shows events relevant to instructions and functional units. Table 3.5 shows events concerning units which are blocked/stalled. Instead of counting the number of events, the number in this table gives the number of cycles for the event type. Table 3.6 shows the events concerning rates and ratios. Indirect events are given in italics.

.

[1] It seems, that on CRAY T3E's the additional logic built around the L2-cache (E-registers, back-map, stream buffers) may lead to wrong L2-cache numbers.
[2] read Processor Cycle Counter
[3] read Processor Cycle Counter
[4] read Tick Counter
[5] read Time Stamp Counter
[6] read Time Stamp Counter
[7] Floating point operations instead of instructions are counted.
[8] See comments on *PE0PE1_30*.
[9] Issued instructions are counted instead of completed instructions.
[10] Integer multiplication and division increments the counter by two
[11] only on Pentium MMX

category	event	event name	DEC Alpha		MIPS	SUN	IBM	Intel	
			21164	21264	R10000	ULTRA	PPC604e	Pentium-MMX	PPro/PII/PIII
1st level cache	read	PCL_L1CACHE_READ							
	write	PCL_L1CACHE_WRITE							
	read or write	PCL_L1CACHE_READWRITE							
	hit	PCL_L1CACHE_HIT							
	miss	PCL_L1CACHE_MISS			$MI1\_9 + MI0\_9$		$IB0\_5 + IB1\_6$		
1st level cache (data)	read	PCL_L1DCACHE_READ				SU0_5		PE0PE1_0	
	write	PCL_L1DCACHE_WRITE				SU0_6		PE0PE1_1	
	read or write	PCL_L1DCACHE_READWRITE	AL1_14						
	hit	PCL_L1DCACHE_HIT	$AL1\_14 - AL2\_5$						
	miss	PCL_L1DCACHE_MISS	AL2_5		MI1_9	SU0_11	IB1_6	PE0PE1_37	PP0PP1_1
1st level cache (instruction)	read	PCL_L1ICACHE_READ						PE0PE1_12	PP0PP1_5
	write	PCL_L1ICACHE_WRITE							
	read or write	PCL_L1ICACHE_READWRITE	AL1_13			SU0_4			
	hit	PCL_L1ICACHE_HIT	$AL1\_13 - AL2\_3$			SU1_4			
	miss	PCL_L1ICACHE_MISS	AL2_3		MI0_9	$SU0\_4 - SU1\_4$	IB0_5	PE0PE1_14	PP0PP1_6

Table 3.1: 1st level cache

category	event	event name	DEC Alpha		MIPS	SUN	IBM	Intel	
			21164	21264	R10000	ULTRA	PPC604e	Pentium-MMX	PPro/PII/PIII
2nd level cache	read	PCL_L2CACHE_READ	AL1_16						
	write	PCL_L2CACHE_WRITE	AL1_17						
	read or write	PCL_L2CACHE_READWRITE	AL1_15[1]			SU0_8			PP0PP1_17
	hit	PCL_L2CACHE_HIT	$AL1\_15 - AL2\_14$			SU1_8			
	miss	PCL_L2CACHE_MISS	AL2_14		$MI1\_10 + MI0\_10$	SU1_9			PP0PP1_13
2nd level cache (data)	read	PCL_L2DCACHE_READ							PP0PP1_11
	write	PCL_L2DCACHE_WRITE							PP0PP1_12
	read or write	PCL_L2DCACHE_READWRITE							$PP0PP1\_11 + PP0PP1\_12$
	hit	PCL_L2DCACHE_HIT							
	miss	PCL_L2DCACHE_MISS			MI1_10				
2nd level cache (instruction)	read	PCL_L2ICACHE_READ							
	write	PCL_L2ICACHE_WRITE							
	read/write	PCL_L2ICACHE_READWRITE							
	hit	PCL_L2ICACHE_HIT							
	miss	PCL_L2ICACHE_MISS			MI0_10				

Table 3.2: Level-2-Cache

category	event	event name	DEC Alpha		MIPS	SUN	IBM	Intel	
			21164	21264	R10000	ULTRA	PPC604e	Pentium-MMX	PPro/PII/PIII
TLB	hit	PCL_TLB_HIT							
	miss	PCL_TLB_MISS			MI1_7		$IB1\_7 + IB0\_6$		
TLB	hit	PCL_ITLB_HIT							
(instruction)	miss	PCL_ITLB_MISS	AL2_4	AL264_1_5			IB1_7	PE0PE1_13	PP0PP1_7
TLB	hit	PCL_DTLB_HIT							
(data)	miss	PCL_DTLB_MISS	AL2_6				IB0_6	PE0PE1_2	

Table 3.3: Transfer-Look-aside-Buffer

category	event	event name	DEC Alpha 21164	DEC Alpha 21264	MIPS R10000	SUN ULTRA	IBM PPC604e	Intel Pentium-MMX	Intel PPro/PII/PIII
cycles		PCL_CYCLES	AL0_0	AL264_0_0	MI0_0	SU0_0	IB3_1	PE0_4	PP0PP1_61
	elapsed cycles	PCL_ELAPSED_CYCLES	PCC[2]	PCC[3]		TC[4]		TSC[5]	TSC[6]
completed	integer	PCL_INTEGER_INSTR	AL1_9				IB0_14		
instructions	floating-point	PCL_FP_INSTR	AL1_10[7]		MI1_5		IB0_15	PE0PE1_30[8]	PP0_0
	load	PCL_LOAD_INSTR	AL1_11		MI1_2		IB0_16		
	store	PCL_STORE_INSTR	AL1_12		MI1_3				
	load or store	PCL_LOADSTORE_INSTR						PE0PE1_36	PP0PP1_0
	sum	PCL_INSTR	AL0_1[9]	AL264_0_1	MI0_15[10]	SU1_1	IB0_2	PE0PE1_20	PP0PP1_44
branch	succ. predicted	PCL_JUMP_SUCCESS			$MI0\_6 - MI1\_8$			PE1_4[11]	PP0PP1_52
instructions	wrong predicted	PCL_JUMP_UNSUCCESS	AL2_2	AL264_1_2	MI1_8			$PE0PE1\_16 - PE1\_4$	PP0PP1_51
	sum	PCL_JUMP		AL264_1_1	MI0_6		IB1_16	PE0PE1_16	PP0PP1_50
atomic	with success	PCL_ATOMIC_SUCCESS	AL2_13		$MI1\_4 - MI0\_5$		IB1_9		
instructions	without success	PCL_ATOMIC_UNSUCCESS			MI0_5				
	sum	PCL_ATOMIC			MI1_4				

Table 3.4: Instructions and functional units

category	event	event name	DEC Alpha		MIPS	SUN	IBM	Intel	
			21164	21264	R10000	ULTRA	PPC604e	Pentium-MMX	PPro/PII/PIII
blocked	integer	PCL_STALL_INTEGER							
functional	floating-point	PCL_STALL_FP					IB2_19		
units	branch	PCL_STALL_JUMP					IB2_12		
	load	PCL_STALL_LOAD						PE0PE1_24	
	store	PCL_STALL_STORE						PE0PE1_23	
	sum	PCL_STALL							PP0PP1_58

Table 3.5: Blocked units

event	event name	DEC Alpha		MIPS	SUN	IBM	Intel	
		21164	21264	R10000	ULTRA	PPC604e	Pentium-MMX	PPro/PII/PIII
MFLOPS	PCL_MFLOPS	$AL1\_10/AL2\_11 * Mhz$		$MI1\_5/MI0\_0 * Mhz$		$IB0\_15/IB1\_1 * Mhz$	$PE0PE1\_30/PE0\_4 * Mhz$	$PP0\_0/PP0PP1\_61 * Mhz$
instr./sec	PCL_IPC	$AL0\_1/AL2\_11$			$SU0\_0/SU1\_0$	$IB0\_15/IB1\_1$	$PE0PE1\_20/PE0\_4$	$PP0PP1\_44/PP0PP1\_61$
L1 Dcache missrate	PCL_L1DCACHE_MISSRATE	$AL2\_5/AL1\_14$					$PE0PE1\_37/PE0PE1\_36$	$PP0PP1\_1/PP0PP1\_0$
L2 Dcache missrate	PCL_L2DCACHE_MISSRATE				$SU1\_9/SU0\_11$			
memory-ops/FP-ops	PCL_MEM_FP_RATIO						$PE0PE1\_36/PE0PE1\_30$	

Table 3.6: Rates and Ratios

3.2 Interface Functions

The interface functions to control the performance counters are given below. All functions are callable from C, C++, Fortran, and Java. All functions return status codes with the following meaning:

PCL_SUCCESS function successful finished

PCL_NOT_SUPPORTED requested event is not supported on this hardware

PCL_TOO_MANY_EVENTS more events requested than performance counters are available

PCL_TOO_MANY_NESTINGS there are more nested calls than allowed (*PCL_MAX_NESTING_LEVEL*)

PCL_TOO_ILL_NESTING either a different number or different types of events are requested in nested calls

PCL_ILL_EVENT event identifier illegal

PCL_MODE_NOT_SUPPORTED performance counting for that mode is not supported

PCL_FAILURE failure for some unspecified reason

3.2.1 PCLquery

With this function, queries are done if a certain functionality is available on this machine. The user supplies in *counter_list* an array of size *ncounter* of event names (of type integers). Event names are any of those introduced in the tables 3.1 to 3.5 in the last section. In *mode*, the user specifies the execution mode for which performance data should be gathered: *PCL_MODE_USER* specifies counting in user mode, *PCL_MODE_SYSTEM* specifies counting in system mode, and *PCL_MODE_USER_SYSTEM* specifies either of both modes. The function returns *PCL_SUCCESS* if the requested functionality is possible (i.e. if the requested events can be counted in parallel), otherwise an error code is returned why the requested events are not supported on this system. No resources are allocated on this call.

```
int PCLquery(
                int             *counter_list,   /* I: requested event counters */
                int             ncounter,        /* I: number of counters */
                unsigned int    mode             /* I: mode flags (PCL_MODE_xxx) */
                );
```

3.2.2 PCLstart

With *PCLstart*, performance counting is started (if it is possible). The user supplies in *counter_list* an array of size *ncounter* of event names. Event names are any of those introduced in the tables 3.1 to 3.5 in the last section. *mode* has the same meaning as in the description of *PCLquery*. If the requested functionality is available, the appropriate performance counters are cleared and started. On success, *PCL_SUCCESS* is returned, otherwise an error code is returned.

```
int PCLstart(
              int             *counter_list,   /* I: events to be counted */
              int             ncounter,        /* I: number of counters */
              unsigned int    mode             /* I: mode flags (PCL_MODE_xxx) */
              );
```

3.2.3 PCLread

Reads out performance counters and returns counter values. Each of the the result values is either written into the (user supplied) integer-typed buffer *i_results_list* or into the (user supplied) floating point typed buffer *fp_results_list* both of size *ncounter*. *PCL_CNT_TYPE* is a 64-bit integer type, *PCL_FP_CNT_TYPE* is a 64-bit floating point type. Which of the buffers is used for the i-th result depends on the requested i-th event type. If the i-th event type is less than PCL_MFLOPS, the result is an integer value which is stored in *i_results_list[i]*. If the i-th event type is greater than or equal to PCL_MFLOPS (i.e. belongs to the category *rates and ratios*), the result is a floating point value stored in *fp_results_list[i]*. If the i-th result is stored in *i_results_list[i]*, the content of *fp_results_list[i]* is undefined, and the same holds for the other way.

Processor	OS	software used	counters saved on context switches
Alpha 21164	Digital Unix 4.0x		yes [14]
Alpha 21264	Digital Unix 4.0e		yes [15]
Alpha 21164	CRAY Unicos/mk		not necessary [16]
R10000	SGI IRIX 6.x		yes
UltraSPARC I/II	Solaris 2.x	perfmon	no
PowerPC 604e	AIX 4.1, 4.2	PMapi	yes
Pentium/PPro/Pentium II/Pentium III	Linux 2.x	msr	no

Table 3.7: Supported systems

The arguments supplied with the call to *PCLread* must correspond to the latest call to *PCLstart*, i.e. the number of requested performance counters must be equal. If no error occurs, *PCL_SUCCESS* is returned, otherwise an error code. The performance counters are (logically) not stopped.

```
int PCLread(
        PCL_CNT_TYPE *      i_result_list,   /* O: int counter values */
        PCL_FP_CNT_TYPE *   fp_result_list,  /* O: fp counter values */
        int                 ncounter         /* I: number of events */
        );
```

3.2.4 PCLstop

Stops performance counting and returns counter values. Result values are written into the (user supplied) buffers *i_result_list* or *fp_result_list* both of size *ncounter*. See *PCLread* for a description how the results are stored in the two arrays. The arguments supplied with the call to *PCLstop* must correspond to the latest call to *PCLstart*, i.e. the number of requested performance counters must be equal. If no error occurs, *PCL_SUCCESS* is returned, otherwise an error code.

```
int PCLstop(
        PCL_CNT_TYPE *      i_result_list,   /* O: int counter values */
        PCL_FP_CNT_TYPE *   fp_result_list,  /* O: fp counter values */
        int                 ncounter         /* I: number of events */
        );
```

3.3 Programming Aspects

The allowed calling sequence is one call to *PCLstart* followed by zero or more calls to *PCLread* followed by one call to *PCLstop*. Between a call to *PCLstart* and *PCLstop* (and possible calls to *PCLread*) may be nested calls to other allowed calling sequences with the same number of events and the same event types.

On system with virtual (low level) performance counters, migrating a process to another processor is possible (SGI, AIX). On the other systems, we bind the executing process to a processor (DEC, SOLARIS)[12], or the process can not migrate (CRAY). On Solaris systems, if the process is not bound to a specific processor, the process gets bound to the processor 0 when executing the *PCLstart* function. On DEC systems, the process gets bound to the processor the process is currently running on.

Currently, performance counters are not saved on context switches on Solaris and Linux systems by our library and therefore performance measurements should be done only on a lightly loaded system.

Currently, we do not check if any other process uses the performance counters as well[13]. Therefore, on certain systems if two distinct processes use performance counters in parallel, they may disturb each other.

To avoid overflow e.g. on systems with 32-bit hardware counters, an interval timer is called on these systems (Solaris, AIX, Linux) which interrupts the process every second. Programs which use the *setitimer* system call (or the *SIGALRM* signal), may be in conflict with PCL.

3.4 Supported Systems

Currently, the Performance Counter Library is available on the systems listed in table 3.7.

[12] On Linux systems, currently it is not possible to bind a process to a processor.

[13] This may be a program using the performance counters directly, or through a different application interface.

3.5 Examples

3.5.1 Simple Example

Figure 3.1 shows a simple example program how to use the Performance Counter Library. First, the list of requested events (*PCL_LOAD_INSTR* for load instructions, and *PCL_L1DCACHE_MISS* for 1st level data cache misses) is put into the array *counter_list*. With the call to *PCLquery* we test, if it is possible to serve these two requested events simultaneously on the computer system where the program is executed. If this is possible, event counting is started with the call to *PCLstart*. After that follows the code to be measured and a call to *PCLstop* to stop performance counting and to read out the performance counter values. Then, the results are printed.

3.5.2 Example with Nested Calls

Figure 3.2 shows an example how to use nested calls. In this example, for the outer loop as well as for each iteration the number of cycles spent in this code section is measured.

3.5.3 Example in Java

Figure 3.3 shows an example how to use PCL in Java.

[14] Only one process can open the *pfm*-device, but spawned children have access to this device as well.
[15] Only one process can open the *pfm*-device, but spawned children have access to this device as well.
[16] There is no multiprogramming on application nodes.

```
#include <pcl.h>

void do_work()

int main(int argc, char **argv)
{
    int counter_list[2];
    int ncounter, res;
    unsigned int mode;
    PCL_CNT_TYPE i_result_list[2];
    PCL_FP_CNT_TYPE fp_result_list[2];


    /* Define what we want to measure. */
    ncounter = 2;
    counter_list[0] = PCL_CYCLES;
    counter_list[1] = PCL_INSTR;

    /* define count mode */
    mode = PCL_MODE_USER;

    /* Check if this is possible on the machine. */
    if( PCLquery(counter_list, ncounter, mode) != PCL_SUCCESS)
        printf("requested events not possible");

    /* Start performance counting.
       We have checked already the requested functionality
       with PCL_query, so no error check would be necessary. */
    res =  PCLstart(counter_list, ncounter, mode);
    if(res != PCL_SUCCESS)
        printf("something went wrong");

    /* Here comes the work to be measured. */
    do_work();

    /* Stop performance counting and get the counter values. */
    if( PCLstop(i_result_list, fp_result_list, ncounter) != PCL_SUCCESS)
        printf("problems with stopping counters");

    /* print out results */
    printf("%f instructions in %f cycles",
           (double)i_result_list[1], (double)i_result_list[0]);
}
```

Figure 3.1: Example program on how to use PCL

```
#include <pcl.h>

#define NITER 4

int main(int argc, char **argv)
{
    int counter_list[1];
    int ncounter, res, iter;
    unsigned int mode;
    PCL_CNT_TYPE i_all_result_list, i_result_list[NITER];
    PCL_FP_CNT_TYPE fp_all_result_list, fp_result_list[NITER];

    /* Define what we want to measure. */
    ncounter = 1;
    counter_list[0] = PCL_CYCLES;

    /* define count mode */
    mode = PCL_COUNT_USER;

    /* Start performance counting. */
    res =  PCLstart(counter_list, ncounter, mode);

    for(iter = 0; iter < NITER; ++iter)

        /* Start performance counting. */
        res =  PCLstart(counter_list, ncounter, mode);

        /* Here comes the work to be measured. */
        do_work();

        /* Stop performance counting and get counter values. */
        res =  PCLstop(&i_result_list[iter], &fp_result_list[iter], ncounter);


    /* Stop performance counting and get the counter values. */
    res =  PCLstop(&i_all_result_list, &fp_all_result_list, ncounter);

    /* print out results */
    printf("used cycles: %f %f %f %f, total: %f",
           (double)i_result_list[0], (double)i_result_list[1],
           (double)i_result_list[2], (double)i_result_list[3],
           (double)i_all_result_list);
}
```

Figure 3.2: Example program on how to use nested calls to PCL

```
// import PCL class description
import PCL;
public class pcl_jtest
{static final int N = 200;               // matrix dimension
  static double[][] a = new double[N][N];
  static double[][] b = new double[N][N];
  static double[][] c = new double[N][N];

  // test method
  static void matadd(double[][] a, double[][] b, double[][] c)
  {int i, j;
    for (i = 0; i < N; ++i)
      for (j = 0; j < N; ++j)
        a[i][j] = b[i][j] + c[i][j];
  }

  // main program
  public static void main(String[] args)
  {int event;
    PCL pcl = new PCL();                  // instantiate PCL
    int mode = pcl.PCL_MODE_USER_SYSTEM; // count mode
    int[] events = new int[1];            // events; array required
    long[] i_result = new long[1];        // int results; array required
    double[] fp_result = new double[1];  // fp results

    // test supported events
    for(event = 0; event < pcl.PCL_MAX_EVENT; ++event)
      {events[0] = event;
        if( pcl.PCLquery(events, 1, mode) == pcl.PCL_SUCCESS)
          {// start counting
            if( pcl.PCLstart(events,1,mode) != pcl.PCL_SUCCESS)
                System.out.println("problem with starting event");

            // test program
            matadd(a,b,c);

            // stop counting
            if( pcl.PCLstop(i_result,fp_result,1) != pcl.PCL_SUCCESS)
                System.out.println("problem with stopping event");

            // print result for event i
            if(event < pcl.PCL_MFLOPS)
              // integer result
              System.out.println(pcl.PCLeventname(i)+":"+i_result[0]);
            else
              // floating point result
              System.out.println(pcl.PCLeventname(i)+":"+fp_result[0]);
          }
      }
   }
}
```

Figure 3.3: Example program in Java.

Chapter 4

Related Projects

In the Parallel Tools Consortium there is a subproject defined called PerfAPI. Its main aspect is to define an API to access all system specific hardware performance counters, i.e. to start/read out/stop all hardware performance counters on a microprocessor with all events available on that system. This is a different approach than ours as we focus on a single framework on all systems, i.e. a uniform application interface as well as a well-defined set of events accessible with uniform names on all systems. For the PerfAPI project, have a look at http://www.cs.utk.edu/ mucci/pdsa/.

There are a lot of interfaces to access performance counters on one specific system, e.g. *libperfex* on SGI systems with the R10000-processor or the *pfm*-device on Digital Unix systems (21064 or 21164 processors). To establish a common platform for performance counting on all POWER and PowerPC microprocessors, IBM has defined an application interface called PMapi. Their approach is as well, to define the set of possible events as the union of all possible events on all POWER and PowerPC microprocessors. On Linux systems, libpperf supports all Pentium, PentiumPro, and Pentium II processors through a common interface.

Chapter 5

Summary

PCL – the Performance Counter Library – is a common interface for portable performance counting on modern microprocessors. It is intended to be used by the expert application programmer who wishes to do detailed analysis on program performance, and it is intended to be used by tool writers who need a common platform to base their work on.

The application interface supports query for functionality, start and stop of performance counting and reading out the values of the performance counters. Nested calls to the functions are possible (with the same events) therefore allowing to do hierarchical performance measurements on sections and subsections of a program. Further, performance counting in user mode, system, and user-or-system mode can be distinguished. Language bindings are available for C, C++, Fortran, and Java.

PCL is available at http://www.fz-juelich.de/zam/PCL/.

Chapter 6

Acknowledgments

We would like to thank the people who have written the software we based our work on. Namely, Richard Enbody for perfmon on UltraSPARC-systems, and M. Patrick Goda and Michael S. Warren for libpperf which itself is based on the msr device implemented by Stephan Meyer on Linux version 2.0.x, 2.1.x, and 2.2.x.

Bibliography

[1] Kirk W. Cameron and Yong Luo. Performance evaluation using hardware performance counters. http://www.c3.lanl.gov/ kirk/isca99/.

[2] Digital Equipment Corporation, Maynard, Massachusetts. *man 7 pfm*.

[3] Digital Equipment Corporation, Maynard, Massachusetts. *Alpha AXP Architecture Handbook*, version 2 edition, 1994.

[4] Silicon Graphics Inc. *man libperfex*.

[5] MIPS Technologies Inc., Mountain View, California. *Definition of MIPS R12000 Performance-counter*.

[6] Marco Zagha and et.al. Performance Analysis using the MIPS R10000 Performance Counters. In *Supercomputing 96*. IEEE Computer Society, 1996.

[7] Sun Microsystems, Palo Alto, California. *UtraSPARC User's Manual*, 1997.

[8] SPARC International, Inc. *The SPARC Architecture Manual, Version 9*, 1997.

[9] Motorala Inc., IBM. *The PowerPC Family : The Bus Interface for 32-Bit Microprocessors*, 3 1997.

[10] James E. Smith Shlomo Weiss. *POWER and PowerPC*. Morgan Kaufmann Publishers, Inc., 1994.

[11] Motorola Inc., IBM. *PowerPC 604e RISC Microprocessor User's Manual*, 3 1998.

[12] *http://developer.intel.com/drg/mmx/AppNotes/perfmon.htm*.

[13] Intel Corp. *Pentium Pro Family Developers Manual 1-3*, 1997.

Appendix A

Performance Counters on Microprocessors

This chapter introduces performance counting aspects of commonly used microprocessors. Each section introduces a microprocessor family and is divided into three subsections: base information on the microprocessor, performance counter events sorted by each performance counter, and in the third subsection additional comments and references to existing implementations to access the performance counters on that specific microprocessor. The second part of each section, the description of the performance counters, is given for each event as follows. The first line contains an internal identifier (2 letters corresponding to the name of the microprocessor), the number of the performance counter, and after a underline another number giving the event number. We will refer to the whole name as a unique identifier in subsequent chapters. The next line contains a manufacturer-specific name or definition (in italics) of the event as found in the manufacturer's literature. After that, a description of the event follows.

A.1 DEC Alpha

To use performance counters on DEC Alpha microprocessors, additional software support is necessary as the low-level interface is given in PAL-Code. Tru64 (formely Digital Unix) has the pseudo device *pfm* [2] which has a high-level interface based on *ioctl*-calls to access the performance counters. The *pfm*-device on systems distinguishes between user and system mode event counting. Only one process per CPU can open the device, but child processes can be spawned which influence the performance counters as well.

On the CRAY T3E, which uses the 21164 microprocessor too, there is no software interface published to access the performance counters.

A.1.1 DEC Alpha 21164

The RISC-processor DEC Alpha 21164 has 3 performance counters. First, let's have a closer look at the architecture of the microprocessor. The first level of caches contain an instruction (*ICACHE*) and a data cache (*DCACHE*), each having a size of 8 KB. The second level cache (*SCACHE*) has a size of 96 KB buffering instructions and data. An additional option is an external third level cache (*BCACHE*). The memory hierarchy is given in figure A.1. A detailed description of the Alpha architecture can be found in [3].

The 21164 contains pipelines of the following types:

- 7-stage integer pipelines
- 9-stage floating point pipelines
- 13-stage memory reference pipeline

The performance counter part on the DEC Alpha 21164 contains 3 counters with distinct purposes. Roughly speaking, counter 0 counts machine cycles or issued instructions, counter 1 counts successful operations, and counter 2 counts unsuccessful operations. For the counters, 2, 24, and 23 different events are defined, respectively, and the counters can operate in parallel. There is one restriction that when counting certain events on counter 2, counter 1 gathers special events.

Events countable on the DEC Alpha 21164 are:

- **Counter 0:**

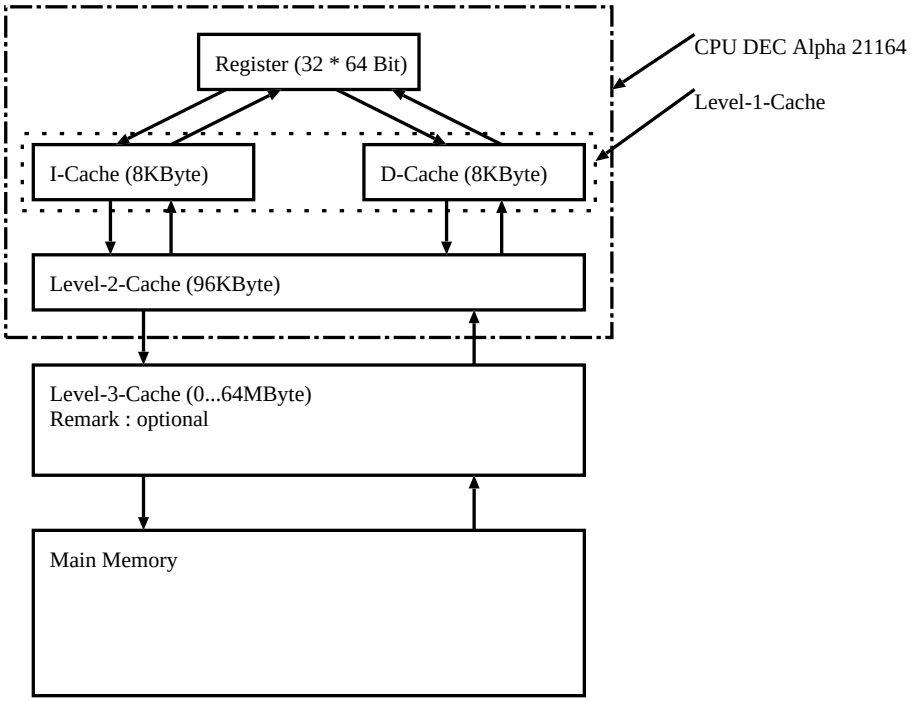

Figure A.1: Principal memory architecture of the DEC Alpha 21164

- AL0_0
  CYCLES
  machine cycles
- AL0_1
  ISSUES
  issued instructions

- **Counter 1:**
  - AL1_0
    NON_ISSUE_CYCLES
    Either no instructions have been issued to the pipeline in the number of cycles, or the pipeline has been stalled for that number of cycles.
  - AL1_1
    SPLIT_ISSUE_CYCLES
    Not all startable instructions have been included into the instruction pipeline.
  - AL1_2
    PIPELINE_DRY
    A parallel execution of instructions was not possible.
  - AL1_3
    REPLAY_TRAP
    If a started instruction could not be further processed, the instruction is issued again in the instruction pipeline, which is called a replay trap.
  - AL1_4
    SINGLE_ISSUE_CYCLES
    Exactly 1 instruction was issued in a cycle.
  - AL1_5
    DUAL_ISSUE_CYCLES
    Exactly 2 instructions were issued in a cycle.
  - AL1_6
    TRIPLE_ISSUE_CYCLES
    Exactly 3 instructions were issued in a cycle.

- AL1_7
  QUAD_ISSUE_CYCLES
  Exactly 4 instructions were issued in a cycle.
- AL1_8
  FLOW_CHANGE
  A jump instruction was executed. Conditional and unconditional jumps are distinguished.
  Remark:
  - If counter 3 counts branch-mispredictions, then branches are counted.
  - If counter 3 counts pc-mispredictions, then jsr (subroutine calls, returns) are counted.
- AL1_9
  INTEGER_OPERATE
  Executed operations in the integer pipelines.
- AL1_10
  FP_INSTRUCTIONS
  Executed operations in the floating point pipelines.
- AL1_11
  LOAD_INSTRUCTIONS
  Executed load instructions.
- AL1_12
  STORE_INSTRUCTIONS
  Executes store instructions.
- AL1_13
  ICACHE_ACCESS
  Accesses to the 1st level instruction cache (ICACHE).
- AL1_14
  DCACHE_ACCESS
  Accesses to the 1st level data cache (DCACHE).
- AL1_15-AL1_21
  "CBOX1"
  Accesses to 2nd or 3rd level cache. There need to be defined additional options [3]:
  - AL1_15
    SCACHE_ACCESS
    Accesses to 2nd level cache (SCACHE).
  - AL1_16
    SCACHE_READ
    Read accesses to 2nd level cache (SCACHE).
  - AL1_17
    SCACHE_WRITE
    Write accesses to 2nd level cache (SCACHE).
  - AL1_18
    SCACHE_VICTIM
    Number of non-completed memory frees in 2nd level cache (SCACHE).
  - AL1_19
    BCACHE_HIT
    Hits in 3rd level cache (BCACHE).
  - AL1_20
    BCACHE_VICTIM
    Number of non-completed memory frees in 3rd level cache (SCACHE).
  - AL1_21
    SYS_REQ
    Requests of additional hardware (multiprocessor system).

- **Counter 2:**
  - AL2_0
    LONG_STALLS
    Number of events that instruction pipeline was blocked for more than 12 cycles.

- AL2_1
  PC_MISPR
  Program counter mispredictions.
- AL2_2
  BRANCH_MISPREDICTS
  Branch mispredictions.
- AL2_3
  ICACHE_MISSES
  Misses in the 1st level instruction cache (ICACHE).
- AL2_4
  ITB_MISSES
  Misses in instruction TLB.
- AL2_5
  DCACHE_MISSES
  Misses in 1nd level data cache (DCACHE).
- AL2_6
  DTB_MISS
  Misses in data TLB.
- AL2_7
  LOADS_MERGED
  An entry in the Miss-Address-File corresponds to a memory request.
- AL2_8
  LDU_REPLAYS
  A replay trap was triggered by a missed load operation.
- AL2_9
  WB_MAF_FULL_REPLAYS
  A replay trap was triggered by a missed write-back operation or by an inconsistency in the miss-address-file.
- AL2_10
  EXTERNAL
  A signal change at the pin "*perf_mon_h*" occurred.
- AL2_11
  CYCLES
  Number of cycles.
- AL2_12
  MEM_BARRIER
  Executed memory barrier instructions.
- AL2_13
  LOAD_LOCKED
  A locked load instruction was executed.
- AL2_14-AL2_21
  "CBOX2"
  Accesses to 2nd or 3rd level cache. There need to be defined additional options [3]:
  - AL2_14
    SCACHE_MISS
    Misses on 2nd level cache.
  - AL2_15
    SCACHE_READ_MISS
    Read misses on 2nd level cache.
  - AL2_16
    SCACHE_WRITE_MISS
    Write misses on 2nd level cache.
  - AL2_17
    SCACHE_SH_WRITE
    Number of write-operations which go to caches other than the processor-specific 2nd level cache.

* AL2_18
  SCACHE_WRITE
  Write accesses to 2nd level cache.
* AL2_19
  BCACHE_MISS
  Misses in 3rd level cache.
* AL2_20
  SYS_INV
  Requests of additional hardware to invalidate a cache line (multiprocessor).
* AL2_21
  SYS_READ_REQ
  Requests of additional hardware to read-copy a cache line (multiprocessor).

A.1.2 DEC Alpha 21264

The DEC Alpha 21264 is a four-way out-of-order-issue microprocessor that performs dynamic scheduling, register renaming, and speculative execution. There are 4 integer execution units and 2 floating-point execution units. The processor includes a 64 KB 1st level instruction cache and a 64 KB 1st level data cache. The 21264 has 2 performance counters of 20 bit width each. Counters 0 is capable of counting one of 2 different events, and counter 1 is capable of counting one of 7 different events. Therefore, the ability to do a detailled performance analysis on the 21264 is significantly reduced compared to the 21164.

Events countable on the DEC Alpha 21264 are:

- **Counter 0:**
  - AL264_0_0
    machine cycles
  - AL264_0_1
    retired instructions
- **Counter 1:**
  - AL264_1_0
    machine cycles
  - AL264_1_1
    retired conditional branches
  - AL264_1_2
    retired branch mispredicts
  - AL264_1_3
    *retired DTB single misses * 2*
  - AL264_1_4
    retired DTB double double misses
  - AL264_1_5
    retired ITB misses
  - AL264_1_6
    retired unaligned traps
  - AL264_1_7
    replay traps

A.2 MIPS R10000/R12000

The microprocessors R10000 and R12000 of MIPS are 64 Bit RISC-microprocessors with integrated performance counters. The differences of the two processors concerning performance counting will be discussed at the end of this section. The R10000 processor has 64 physical registers and 32 logical registers. The 1st level cache is split between a data cache and an instruction cache, both of size 32 KB. The 2nd level cache can be between 512 KB and 16 MB and the cache is a unified buffer at it caches data as well as instructions. The main memory can be up to 1 TB. Figure A.2 shows a picture of the memory architecture of the processor.

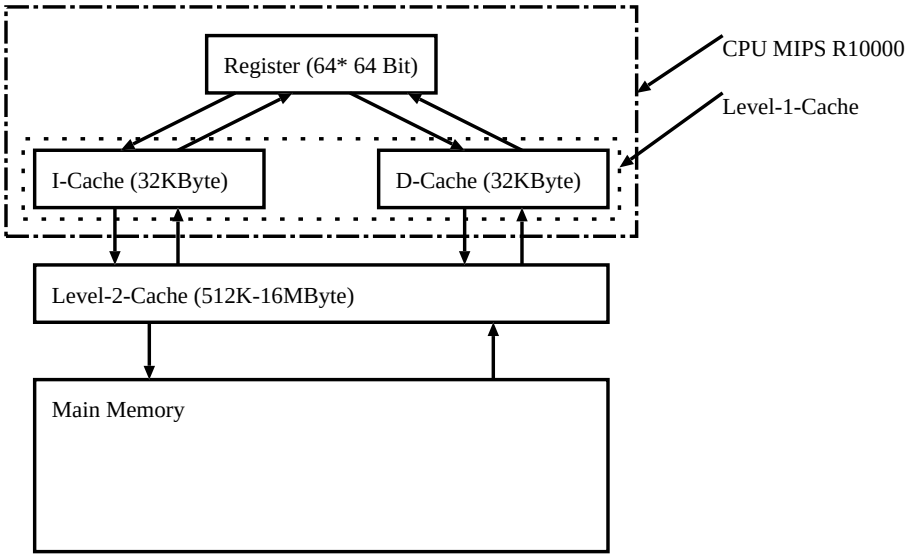

Figure A.2: Memory hierarchy of the MIPS R10000

The R10000 microprocessor has 2 performance counters (a description can be found at http://www.sgi.com/processors/r10k/performance.html) each capable of counting one of 16 different events. The R10000 has 5 execution pipelines executing decoded instructions. There are 2 integer pipelines (*ALU1*, *ALU2*), 2 floating point pipelines (*FPU1*, *FPU2*), and 1 address pipeline (*LOAD/STORE*). The integer and floating point pipelines can operate in parallel. For a better understanding we define the two following terms:

- *issued*: An instruction was decoded and supplied to the executing unit.
- *graduated*: An execution of an instruction has finished and all instruction issued before the instruction have finished, too.

Another term to be defined is *SCTP-Logic* which is the Secondary Cache Transaction Processing Logic, which has the task to store up to 4 internally generated or 1 externally generated 2nd level cache transactions.

- **Counter 0:**
  - MI0_0
    Cycles
    Machine cycles.
  - MI0_1
    Instructions issued
    The counter is incremented with the sum of the following events:
    - integer operations completed at this cycle. There can be 0-2 operations each cycle.
    - floating-point-operations completed at this cycle. There can be 0-2 operations each cycle.
    - load/store operations which have been delivered in the last cycle to the address pipeline. There can be 0 or 1 each cycle.
  - MI0_2
    Load/prefetch/sync/CacheOp issued
    Each of these instructions is counted when started.
  - MI0_3
    Stores(including store-conditional) issued
    Each time a store operations is delivered to the address calculation unit, the counter is incremented.
  - MI0_4
    Store conditional issued
    Each time a conditional store operations is delivered to the address calculation unit, the counter is incremented.

- MI0_5
  Failed store conditional
  The counter is incremented each time a conditional store failed.
- MI0_6
  Conditional Branch resolved
  Count all resolved conditional branches.
- MI0_7
  Quadwords written back from secondary cache
  Counter is incremented each time a quad-word is written from the 2nd level cache to the output buffer.
- MI0_8
  Correctable ECC errors on secondary cache data
  A correctable 1-bit ECC error occurred while reading a quadword from the 2nd level cache.
- MI0_9
  Instruction cache misses
  Misses in the instruction cache.
- MI0_10
  Secondary cache misses (instruction)
  Instruction misses in the 2nd level cache.
- MI0_11
  Secondary cache way mispredicted (instruction)
  An attempt was made to load an instruction from the 2nd level cache and the entry is marked as invalid.
- MI0_12
  External intervention requests
  Number of requests to the *SCTP-Logic* from outside of the processor (I/O devices, multiprocessor etc.) for a copy of a cache line marked as *shared*.
- MI0_13
  External invalidate requests
  Number of requests to the *SCTP-Logic* from outside of the processor (I/O devices, multiprocessor etc.) for invalidation of a cache line marked.
- MI0_14
  Functional unit completion cycles
  The counter is incremented if at least one of the functional units has completed an operations in this cycle.
- MI0_15
  Instruction graduated
  The counter is incremented with the number of instructions which have been completed in the last cycle. An integer multiplication or division increments the counter by 2.

- **Counter 1:**
  - MI1_0
    Cycles
    Machine cycles.
  - MI1_1
    Instructions graduated
    The counter is incremented by the number of instructions which have been completed in the last cycle. An integer multiplication and division increments by 2.
  - MI1_2
    Load/prefetch/sync/CacheOp graduated
    Every completed instruction of this type is counted.
  - MI1_3
    Stores (including store-conditionals) graduated
    Every completed store operation is counted.

- MI1_4
  Store conditionals graduated
  Every conditional store is counted independently of success. This is possible at most once a cycle.
- MI1_5
  Floating-point instructions graduated
  Floating point instructions completed in the last cycle (0-4 each cycle).
- MI1_6
  Quadwords written back from primary cache
  The counter is incremented by 1, if in a cycle at least one quadword is written back from the 1st level cache to the 2nd level cache.
- MI1_7
  TLB refill exceptions
  TLB misses are counted in the cycle after they occur.
- MI1_8
  Branches mispredicted
  The counter is incremented on every mispredicted branch.
- MI1_9
  Primary data cache misses
  Miss in the primary data cache.
- MI1_10
  Secondary cache misses (data)
  Miss in the secondary cache caused by a data access.
- MI1_11
  Secondary cache way mispredicted (data)
  The counter is incremented if the 2nd level cache controller tries to access the 2nd level cache after a previous access failed.
- MI1_12
  External intervention request is determined to have hit in secondary cache
  The processor got an external request for a copy of a 2nd level cache block.
- MI1_13
  External invalidate request is determined to have hit in secondary cache
  The processor got an external request to invalidate a 2nd level cache block.
- MI1_14
  Stores/prefetches with store hint to CleanExklusive secondary cache blocks
  The SCTP-logic got a request for status change of a cache line from *CleanExclusive* to *DirtyExklusive*.
- MI1_15
  Stores/prefetches with store hint to Shared secondary cache blocks
  The status of a cache line was changed from *Shared* to *DirtyExklusive*.

Software support for the performance counters on R10000 processors is available either on a lower level in IRIX 6.x through the */proc* file system or on a higher level through the *perfex* library [4]. The kernel maintains data structures for 32 virtual performance counters with a size of 64 bits each. It is possible to distinguish between counting in user mode, system mode, or both. When running in user mode, performance counters are saved on context switches. For the *perfex* library, the routine *start_counters* zeroes out the internal counters, and *read_counters* stops the counters after reading them.

Different to the R10000, the R12000 has 4 counters each capable of counting one of 32 events. For counter 1, a trigger mechanism was included such that an event is counted by counter 1 if any of the other counters reached a certain value. Additionally, conditional counting is possible. For example, it is possible to count the number of cycles in which 4 instructions have been completed. Also, some semantic inaccuracies concerning the definition of events have been clarified [5]. An introduction to measurement and interpretation of events can be found in [6].

A.3 SUN ULTRASparc I/II

The UltraSPARC I/II 64-bit microprocessors of SUN have the possibility to count performance relevant events. A detailed description of the SPARC V9 architecture can be found in [7]. Both variants have

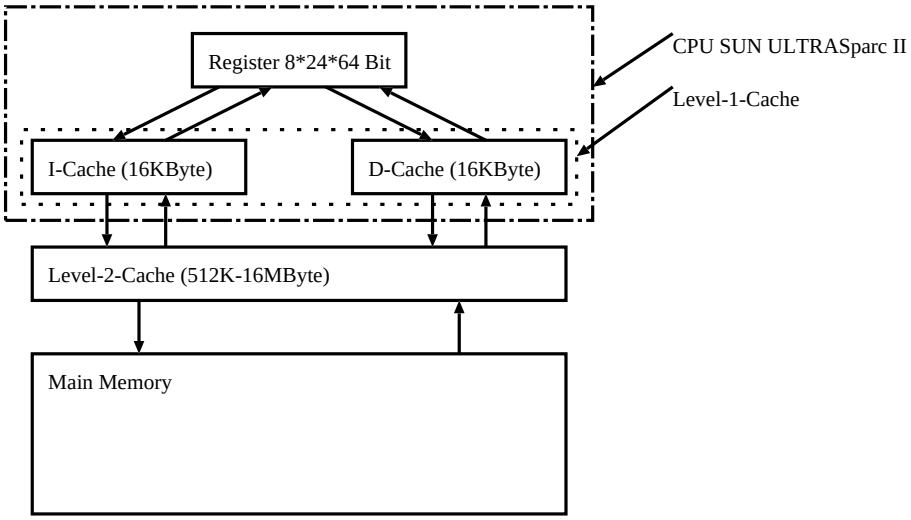

Figure A.3: Memory hierarchy of the SUN ULTRASparc II

8 times 24 64-bit registers which are organized in so-called windows to optimize argument passing on subroutine calls without time-consuming copying of registers to memory. The 1st level cache has a 16 KB data (D-cache) and a 16 KB instruction cache (I-Cache). The 2nd level cache (E-cache) has a size of 512 KB up to 4 MB on UltraSPARC I, and 512 KB up to 16 MB on UltraSPARC II. The main memory can be as large as 2 TB (see figure A.3).

Another important component of the supporting logic is the *UPA*, the Universal Port Architecture, which connects several processors over a high-speed crossbar-switch.

The microprocessor contains two performance counters (PIC0, PIC1), which are able to count different events. Each counter can count one of 12 different events, two events can be counted on both counters, which sums up to a total of 22 different events [8]. Additionally, there exists a elapsed cycle counter.

- **Counter PIC0:**
  - SU0_0
    Cycle_cnt
    Machine cycles.
  - SU0_1
    Instr_cnt
    Instructions graduated.
  - SU0_2
    Dispatch0_IC_miss
    Number of cycles waiting after a miss in the 1st level instruction cache (including handling of a follow-on E-cache miss).
  - SU0_3
    Dispatch0_storeBuf
    Number of cycles a write buffer could not store new values (next instruction is a store instruction).
  - SU0_4
    IC_ref
    1st level instruction cache references.
  - SU0_5
    DC_rd
    1st level data cache read references.
  - SU0_6
    DC_wr
    1st level data cache write references.
  - SU0_7
    Load_use
    Number of cycles instructions are waiting on a previous load operation.

- SU0_8
  EC_ref
  Number of 2nd level cache references.
- SU0_9
  EC_write_hit_RDO
  Number of hits on 2nd level cache read accesses in a *read for ownership*-UPA-transaction.
- SU0_10
  EC_snoop_inv
  Number of cache line invalidations due to a UPA-transactions.
- SU0_11
  EC_rd_hit
  Number of E-cache read hits caused by 1st level data cache miss.

- **Counter PIC1 counts:**
  - SU1_0
    Cycle_cnt
    Machine cycles.
  - SU1_1
    Instr_cnt
    Instructions graduated.
  - SU1_2
    Dispatch0_mispred
    Number of cycles waiting with an empty instruction buffer after a wrong branch prediction.
  - SU1_3
    Dispatch0_FP_use
    Number of cycles which waits the first instruction in a group because the result of a previous floating-point operation is not available.
  - SU1_4
    IC_hit
    Number of 1st level instruction cache hits.
  - SU1_5
    DC_rd_hit
    Number of 1st level data cache read hits.
  - SU1_6
    DC_wr_hit
    Number of 1st level data cache write hits.
  - SU1_7
    Load_use_RAW
    Number of cycles load operations spent in the instruction pipeline while at the same time a read-write-inconsistency exists because of a not-completed load operation.
  - SU1_8
    EC_hit
    Number of 2nd level cache hits.
  - SU1_9
    EC_wb
    Number of 2nd level cache misses causing a write-back operation.
  - SU1_10
    EC_snoop_cb
    Number of UPA-transactions which caused a copy-back of a 2nd level cache line.
  - SU1_11
    EC_ic_hit
    Number of 2nd level cache read hits caused by a 1st level instruction cache miss.

The performance registers are controlled by the Performance Control Register (PCR) which can be accessed only in privileged mode. Accesses to the PIC-registers may be either in user or privileged mode, dependent on a bit in the PCR which can be changed in privileged mode. Event counting can be done either

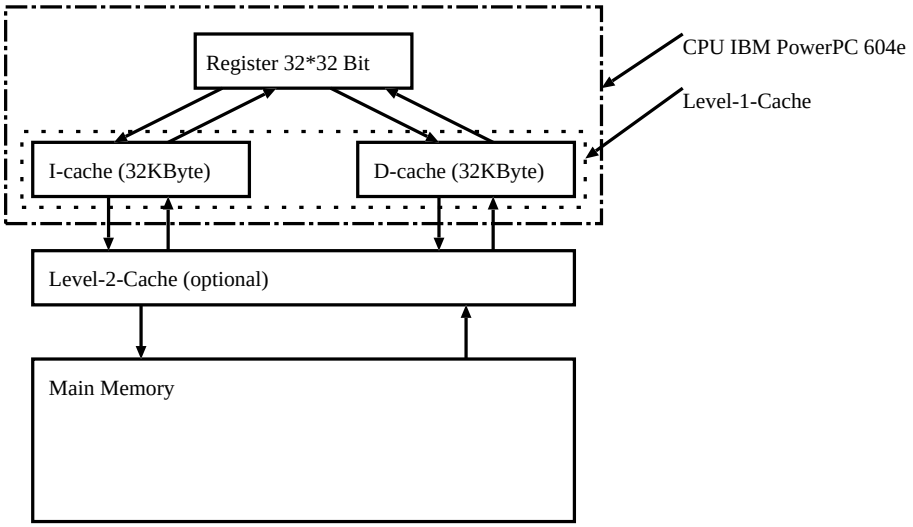

Figure A.4: Memory hierarchy of the IBM PowerPC 604e

for the user mode, system mode, or both. Overflow of the counters is silently. For accurate timing, event counting should be done as taking the difference between two reads of a performance counter.

The actual version 2.6 of the Solaris operating system has not support for the performance counters in form of a programming interface. A software library to access the performance counters in a convenient way is perfmon from Richard Enbody. A drawback of this package is, that neither process migration to another CPU on a multiprocessor machine nor a context switch to another process on the same CPU is handled.

A.4 IBM PowerPC 604e

The PowerPC 604e is a 32-bit microprocessor with 32 32-bit integer and 32 32-bit floating point registers. The 1st level cache consists of a 32 KB data cache (D-cache) and a 32 KB instruction cache (I-cache). Different to other microprocessors, the PowerPC 604e has no on-chip logic to control a 2nd level chip but signals are available for additional cache logic [9]. On figure A.4, the additional logic has been included as most of the non-embedded uses of the PowerPC 604e use a 2nd level cache. Additionally, there exist performance counter events concerning the 2nd level cache. A detailed description of the PowerPC architecture can be found in [10].

The pipelines of the PowerPC 604e consist of:

- a 5-stage branch unit (BPU/CRU)
- a 6-stage integer unit (SCIU1/SCIU2/MCIU)
- a 7-stage load/store unit (LSU)
- an 8-stage floating-point unit (FPU)

Sub-unit names are:

- *BPU* branch prediction unit
- *CRU* control register unit
- *SCIUx* single-cycle integer unit
- *MCIU* multiple-cycle integer unit

The PowerPC 604e has 4 performance counters (PMC1/PMC2/PMC3/PMC4) capable of counting 116 different events [11].

- **Counter PMC1 counts:**

– IB0_0
000 0000 Nothing. Register counter holds current value.
The counter keeps its current value.

– IB0_1
000 0001 Processor cycles 0b1. Count every cycle.
Number of cycles the processor executes ”0b1”.

– IB0_2
000 0010 Number of instructions completed every cycle.
Number of instructions completed each cycle.

– IB0_3
000 0011 RTCSELECT bit transition. 0 = 47, 1 = 51, 2 = 55, 3 = 63 (bits from the time base lower register).
Bit-transitions on the RTCSELECT-Pin.

– IB0_4
000 0100 Number of instructions dispatched.
Number of instructions arrived at the 3rd stage of the instruction pipeline.

– IB0_5
000 0101 Instruction cache misses.
Number of 1st level instruction cache misses.

– IB0_6
000 0110 Data TLB misses (in order).
Number of misses in the translation look-aside buffer for data.

– IB0_7
000 0111 Branch misprediction correction from execute stage.
Number of correctable branch misses in the execution phase of the 4th stage of the pipeline.

– IB0_8
000 1000 Number of reservations requested. The lwarx instruction is ready for execution in the LSU.
Number of reservations for an atomic load instruction in the LSU.

– IB0_9-IB0_10
000 1001 Number of data cache load misses exceeding the threshold value with lateral L2 cache intervention.
000 1010 Number of data cache store misses exceeding the threshold value with lateral L2 cache intervention.
Number of 1st level data cache misses which exceeded a limit value and additionally, L2_INT signal was active.

– IB0_11
000 1011 Number of mtspr instructions dispatched.
Number of *mtspr* instructions arrived at the 3rd stage of the pipeline.

– IB0_12-IB0_15
000 1100 Number of sync instructions completed.
000 1101 Number of eieio instructions completed.
000 1110 Number of integer instructions completed every cycle (no loads or stores).
000 1111 Number of floating-point instructions completed every cycle (no loads or stores).
Number of completed mtspr/sync/eieio/integer/floating-point instructions.

– IB0_16-IB0_18
001 0000 LSU produced result.
001 0001 SCIU1 produced result for an add, subtract, compare, rotate, shift, or logical instruction.
001 0010 FPU produced result.
Number of results generated at the LSU/SCIU1/FPU units.

– IB0_19-IB0_21
001 0011 Number of instructions dispatched to the LSU.
001 0100 Number of instructions dispatched to the SCIU1.
001 0101 Number of instructions dispatched to the FPU.
Number of instructions issued from the 3rd stage of the instruction pipeline to the LSU/SCIU1/FPU unit.

- IB0_22
  001 0110 Valid snoop requests received from outside the 604e. Does not distinguish hits or misses.
  Number of snoop requests.
- IB0_23-IB0_24
  001 0111 Number of data cache load misses exceeding the threshold value without lateral L2 intervention.
  001 1000 Number of data cache store misses exceeding the threshold value without lateral L2 intervention.
  Number of 1st level data cache misses which exceeded a limit value and additionally, L2_INT signal was not active.
- IB0_25-IB0_27
  001 1001 Number of cycles the branch unit is idle.
  001 1010 Number of cycles MCIU0 is idle.
  001 1011 Number of cycles the LSU is idle. No new instructions are executing; however, active loads or stores may be in the queues.
  Number of cycles the BPU/MCIU0/LSU units were idle.
- IB0_28
  001 1100 Number of times the L2_INT is asserted (regardless of TA state).
  Number of times L2_INT signal was asserted.
- IB0_29
  001 1101 Number of unaligned loads.
  Number of unaligned loads.
- IB0_30
  001 1110 Number of entries in the load queue each cycle (maximum of five). Although the load queue has four entries, a load miss latch may hold a load waiting for data from memory.
  Number of load queue entries per cycle (max. of 5).
- IB0_31
  001 1111 Number of instruction breakpoint hits.
  Number of times instructions hit a breakpoint.

- **Counter PMC2 counts:**
  - IB1_0
    00 0000 Nothing. Register counter holds current value.
    The counter keeps its current value.
  - IB1_1
    00 0001 Processor cycles 0b1. Count every cycle.
    Number of cycles the processor executes ”0b1”.
  - IB1_2
    00 0010 Number of instructions completed every cycle.
    Number of instructions completed every cycle.
  - IB1_3
    00 0011 RTCSELECT bit transition. 0 = 47, 1 = 51, 2 = 55, 3 = 63 (bits from the time base lower register).
    Number of bit transitions on the RTCSELECT-pin.
  - IB1_4
    00 0100 Number of instructions dispatched.
    Number of instructions dispatched to the 3rd stage of the instruction pipeline.
  - IB1_5
    00 0101 Number of cycles a load miss takes.
    Number of load miss cycles.
  - IB1_6
    00 0110 Data cache misses (in order).
    Number of 1st level data cache misses.
  - IB1_7
    00 0111 Number of instruction TLB misses.
    Number of misses in the translation look-aside buffer for instructions.

– IB1_8
00 1000 Number of branches completed. Indicates the number of branch instructions being completed every cycle (00 = none, 10 = one, 11 = two, 01 is an illegal value).
Number of completed branch instructions every cycle (max. of 2).

– IB1_9
00 1001 Number of reservations successfully obtained (stwcx. operation completed successfully).
Number of successfully completed atomic store instructions.

– IB1_10
00 1010 Number of mfspr instructions dispatched (in order).
Number of *mfspr*-instructions arrived at the 3rd stage of the instruction pipeline.

– IB1_11
00 1011 Number of icbi instructions. It may not hit in the cache.
Number of *icbi*-instructions without necessary hitting the cache.

– IB1_12
00 1100 Number of pipeline "flushing" instructions (sc, isync, mtspr (XER), mcrxr, floating-point operation with divide by 0 or invalid operand and MSR[FE0, FE1] = 00, branch with MSR[BE] = 1, load string indexed with XER = 0, and SO bit getting set)
Number of instructions flushing the pipeline.

– IB1_13-IB1_15
00 1101 BPU produced result.
00 1110 SCIU0 produced result (of an add, subtract, compare, rotate, shift, or logical instruction).
00 1111 MCIU produced result (of a multiply/divide or SPR instruction).
Number of results produced by the BPU/SCIU0/MCIU-units.

– IB1_16-IB1_17
01 0000 Number of instructions dispatched to the branch unit.
01 0001 Number of instructions dispatched to the SCIU0.
Number of instructions issued from the 3rd stage of the instruction pipeline to the BPU/SCIU0-units.

– IB1_18
01 0010 Number of loads completed. These include all cache operations and tlbie, tlbsync, sync, eieio and icbi instructions.
Number of completed load instructions.

– IB1_19
01 0011 Number of instructions dispatched to the MCIU.
Number of instructions issued from the 3rd stage of the instruction pipeline to the MCIU-unit.

– IB1_20
01 0100 Number of snoop hits occurred.
Number of snoop hits.

– IB1_21
01 0101 Number of cycles during which the MSR[EE] bit is cleared.
Number of cycles during which the MSR[EE] bit is cleared.

– IB1_22-IB1_24
01 0110 Number of cycles the MCIU is idle.
01 0111 Number of cycles SCIU1 is idle.
01 1000 Number of cycles the FPU is idle.
Number of cycles the SCIU1/MCIU/FPU-unit is idle.

– IB1_25
01 1001 Number of cycles the L2_INT signal is active (regardless of TA state).
Number of cycles the L2_INT-pin had an active level.

– IB1_26-IB1_30
01 1010 Number of times four instructions were dispatched.
01 1011 Number of times three instructions were dispatched.
01 1100 Number of times two instructions were dispatched.
01 1101 Number of times one instruction was dispatched.
Number of times 1/2/3/4 instructions arrived at the 3rd stage of the instruction pipeline.

- IB1_31
  01 1110 Number of unaligned stores.
  Number of unaligned stores.
- IB1_32
  01 1111 Number of entries in the store queue each cycle (maximum of six).
  Number of entries in the store-queue every cycle (max. of 6).

- **Counter PMC3 counts:**
  - IB2_0
    0 0000 Nothing. Register counter holds current value.
    The counter keeps its current value.
  - IB2_1
    0 0001 Processor cycles 0b1. Count every cycle.
    Number of cycles the processor executes ”0b1”.
  - IB2_2
    0 0010 Number of instructions completed every cycle.
    Number of instructions completed every cycle.
  - IB2_3
    0 0011 RTCSELECT bit transition. 0 = 47, 1 = 51, 2 = 55, 3 = 63 (bits from the time base lower register).
    Number of bit-transitions on the RTCSELECT-pin.
  - IB2_4
    0 0100 Number of instructions dispatched.
    Number of instructions arrived at the 3rd stage of the instruction pipeline.
  - IB2_5-IB2_7
    0 0101 Number of cycles the LSU stalls due to BIU or cache busy. Counts cycles between when a load or store request is made and a response was expected. For example, when a store is retried, there are four cycles before the same instruction is presented to the cache again. Cycles in between are not counted.
    0 0110 Number of cycles the LSU stalls due to a full store queue.
    0 0111 Number of cycles the LSU stalls due to operands not available in the reservation station.
    Number of cycles the LSU-unit was blocked either because the LSU-unit was busy or the cache was busy or the store queue was full or an operand was not available.
  - IB2_8
    0 1000 Number of instructions written into the load queue. Misaligned loads are split into two transactions with the first part always written into the load queue. If both parts are cache hits, data is returned to the rename registers and the first part is flushed from the load queue. To count the instructions that enter the load queue to stay, the misaligned load hits must be subtracted.
    Number of instructions in the load queue.
  - IB2_9
    0 1001 Number of cycles that completion stalls for a store instruction.
    Number of cycles that completion stalls for a store instruction.
  - IB2_10
    0 1010 Number of cycles that completion stalls for an unfinished instruction.
    Number of cycles that completion stalls for an unfinished instruction.
  - IB2_11
    0 1011 Number of system calls.
    Number of system calls.
  - IB2_12
    0 1100 Number of cycles the BPU stalled as branch waits for its operand.
    Number of cycles the BPU waits for an operand.
  - IB2_13
    0 1101 Number of fetch corrections made at the dispatch stage. Prioritized behind the execute stage.
    Number of fetch corrections made at the 3rd stage of the instruction pipeline.

– IB2_14
0 1110 Number of cycles the dispatch stalls waiting for instructions.
Number of cycles the 1st stage of the instruction pipeline waited for instructions.

– IB2_15
0 1111 Number of cycles the dispatch stalls due to unavailability of reorder buffer (ROB) entry. No ROB entry was available for the first non-dispatched instruction.
Number of cycles the 1st stage of the instruction pipeline waited because the reorder buffer was not available.

– IB2_16
1 0000 Number of cycles the dispatch unit stalls due to no FPR rename buffer available. First non-dispatched instruction required a floating-point reorder buffer and none was available.
Number of cycles the 1st stage of the instruction pipeline waited because the FPR-rename buffer was not available.

– IB2_17-IB2_18
1 0001 Number of instruction table search operations.
1 0010 Number of data table search operations. Completion could result from a page fault or a PTE match.
Number of search operations in the data/instruction table.

– IB2_19-IB2_20
1 0011 Number of cycles the FPU stalled.
1 0100 Number of cycles the SCIU1 stalled.
Number of cycles the FPU-/SCIU1-unit was blocked.

– IB2_21
1 0101 Number of times the BIU forwards non-critical data from the line-fill buffer.
Number of transfers of uncritical data from the line-fill buffer done by the bus-interface unit and initiated by the BIU. to the

– IB2_22
1 0110 Number of data bus transactions completed with pipelining one deep with no additional bus transactions queued behind it.
Number of completed data bus transactions without additional bus transactions queued.

– IB2_23
1 0111 Number of data bus transactions completed with two data bus transactions queued behind.
Number of completed data bus transactions with two additional bus transactions queued.

– IB2_24
1 1000 Counts pairs of back-to-back burst reads streamed without a dead cycle between them in data streaming mode
Number of paired *back-to-back-burst*-read accesses without intervening idle cycles.

– IB2_25
1 1001 Counts non-$\overline{ARTRY}d$ *processor kill transactions caused by a write-hit-on-shared condition*
Number of invalidated cache lines caused by a write hit to a shared line.

– IB2_26
1 1010 This event counts non-$\overline{ARTRY}d$ *write-with-kill address operations that originate from the three castout buffers. These include high-priority write-with-kill transactions caused by a snoop hit on modified data in one of the BIU's three copy-back buffers. When the cache block on a data cache miss is modified, it is queued in one of three copy-back buffers. The miss is serviced before the copy-back buffer is written back to memory as a write-with-kill transaction.*
Number of *Write-with-kill*-address operations.

– IB2_27
1 1011 Number of cycles when exactly two castout buffers are occupied.
Number of cycles when exactly two castout buffers are occupied. *Castout*-buffer are used to write 1st level data cache lines to memory.

– IB2_28
1 1100 Number of data cache accesses retried due to occupied castout buffers.
Number of retried 1st ;level data cache accesses due to occupied castout buffer.

– IB2_29
1 1101 Number of read transactions from load misses brought into the cache in a shared state.
Number of read transactions which (after a miss) brought a 1st level cache line into the cache with a status of *shared*.

– IB2_30
1 1110 CRU Indicates that a CR logical instruction is being finished.
Number of logical instructions completed in the CRU.

- **Counter PMC4 counts:**

  – IB3_0
  0 0000 Nothing. Register counter holds current value.
  The counter keeps its current value.

  – IB3_1
  0 0001 Processor cycles 0b1. Count every cycle.
  Number of cycles the processor executes ”0b1”.

  – IB3_2
  0 0010 Number of instructions completed every cycle.
  Number of instructions every cycle.

  – IB3_4
  0 0011 RTCSELECT bit transition. 0 = 47, 1 = 51, 2 = 55, 3 = 63 (bits from the time base lower register).
  Number of bit-transitions on the RTCSELECT-pin.

  – IB3_5
  0 0100 Number of instructions dispatched.
  Number of instructions arrived at the 3rd stage of the instruction pipeline.

  – IB3_6-IB3_8
  0 0101 Number of cycles the LSU stalls due to busy MMU.
  0 0110 Number of cycles the LSU stalls due to the load queue full.
  0 0111 Number of cycles the LSU stalls due to address collision.
  Number of cycles the LSU stalled because of a busy MMU, full load queue, or address collision.

  – IB3_9
  0 1000 Number of misaligned loads that are cache hits for both the first and second accesses.
  Number of misaligned loads that are cache hits for both the first and second accesses.

  – IB3_10
  0 1001 Number of instructions written into the store queue.
  Number of instructions written into the store queue.

  – IB3_11
  0 1010 Number of cycles that completion stalls for a load instruction.
  Number of cycles the completion of an instructions stalled because of a load instruction.

  – IB3_12
  0 1011 Number of hits in the BTAC. Warning-if decode buffers cannot accept new instructions, the processor re-fetches the same address multiple times.
  Number of hits in the *Branch Target Address Cache*.

  – IB3_13
  0 1100 Number of times the four basic blocks in the completion buffer from which instructions can be retired were used
  Number of times the four basic blocks in the completion buffer from which instructions can be retired were used.

  – IB3_14
  0 1101 Number of fetch corrections made at decode stage.
  Number of corrections made between the 1st and 2nd stage of the instruction pipeline.

  – IB3_15-IB3_18
  0 1110 Number of cycles the dispatch unit stalls due to no unit available. First non-dispatched instruction requires an execution unit that is either full or a previous instruction is being dispatched to that unit.
  0 1111 Number of cycles the dispatch unit stalls due to unavailability of GPR rename buffer.

First non-dispatched instruction requires a GPR reorder buffer and none are available.
1 0000 Number of cycles the dispatch unit stalls due to no CR rename buffer available. First non-dispatched instruction requires a CR rename buffer and none is available.
1 0001 Number of cycles the dispatch unit stalls due to CTR/LR interlock. First non-dispatched instruction could not dispatch due to CTR/LR/mtcrf interlock.
Number of cycles spent at the 3rd stage of the instruction pipeline waiting for any of the conditions:

  * in the 4th stage of the pipeline (MCIU/SCIU0/SCIU1..) was no unit available
  * no GPR-Rename-Buffer was available
  * no CR-Rename-Buffer was available
  * the Counter- or Link-Register was locked

– IB3_19-IB3_20
1 0010 Number of cycles spent doing instruction table search operations.
1 0011 Number of cycles spent doing data table search operations.
Number of cycles spent searching in the data/instruction table.

– IB3_21-IB3_22
1 0100 Number of cycles SCIU0 was stalled.
1 0101 Number of cycles MCIU was stalled.
Number of cycles the MCIU/SCIU0 was stalled.

– IB3_23
1 0110 Number of bus cycles after an internal bus request without a qualified bus grant.
Number of bus-cycles after an internal bus request without a qualified bus grant.

– IB3_24
1 0111 Number of data bus transactions completed with one data bus transaction queued behind
Number of completed data-bus transactions with one data bus transaction queued behind.

– IB3_25
1 1000 Number of write data transactions that have been reordered before a previous read data transaction using the DBWO feature
Number of write data transactions that have been reordered before a previous read data transaction.

– IB3_26
1 1001 Number of $\overline{ARTRY}d$ *processor address bus transactions.*
Number of address bus transactions caused by a signal change at the $\overline{ARTRY}d$-pin.

– IB3_27
1 1010 Number of high-priority snoop pushes. Snoop transactions, except for write-with-kill, that hit modified data in the data cache cause a high-priority write (snoop push) of that modified cache block to memory. This operation has a transaction type of write-with-kill. This event counts the number of non-$\overline{ARTRY}d$ *processor write-with-kill transactions that were caused by a snoop hit on modified data in the data cache. It does not count high-priority write-with-kill transactions caused by snoop hits on modified data in one of the BIU's three copy-back buffers.*
Number of high-priority snoop pushes.

– IB3_28-IB3_29
1 1011 Number of cycles for which exactly one castout buffer is occupied
1 1100 Number of cycles for which exactly three castout buffers are occupied
Number of cycles for which exactly one/three castout buffer is/are occupied.

– IB3_30
1 1101 Number of read transactions from load misses brought into the cache in an exclusive (E) state
Number of read transactions caused by a load miss and which got brought into the cache in exclusive state.

– IB3_31
1 1110 Number of un-dispatched instructions beyond branch
Number of undispatched instructions beyond branch.

IBM has the PMapi library which supports access to the performance counters on different PowerPC and POWER chips. PMapi supports the distinction between supervisor mode, problem (user) mode, or both. On AIX versions 4.2 and higher, performance counter status is saved and restored on context switches.

A.5 Intel Pentium Family

A.5.1 Intel Pentium

The Intel Pentium is a 32-bit CISC microprocessor. The Pentium has 2 performance counters with most of the events countable by either of the counters and only some events countable only by a specific counter (as noted). With the introduction of the MMX-extensions, Pentium's with MMX have defined more events as stated (*MMX-extensions*). We have left out all events which are specific to the MMX functional unit as compilers normally do not generate code for this unit.

The events countable by both counters are:

- PE0PE1_0
  00H DATA_READ
  Number of memory data read operations.
- PE0PE1_1
  01H DATA_WRITE
  Number of memory data write operations.
- PE0PE1_2
  02H DATA_TLB_MISS
  Number of misses to the data cache translation look-aside buffer.
- PE0PE1_3
  03H DATA_READ_MISS
  Number of memory read accesses that miss the internal data cache.
- PE0PE1_4
  04H DATA_WRITE_MISS
  Number of memory write accesses that miss the internal data cache.
- PE0PE1_5
  05H WRITE_HIT_TO_M-_OR_E-STATE_LINES
  Number of write hits to exclusive or modified lines in the data cache.
- PE0PE1_6
  06H DATA_CACHE_LINES_WRITTEN_BACK
  Number of dirty lines that are written back.
- PE0PE1_7
  07H EXTERNAL_SNOOPS
  Number of accepted external snoops.
- PE0PE1_8
  08H EXTERNAL_DATA_CACHE_SNOOP_HITS
  Number of external snoops to the data cache.
- PE0PE1_9
  09H MEMORY ACCESSES IN BOTH PIPES
  Number of data memory reads or writes that are paired in both pipes of the pipeline.
- PE0PE1_10
  0AH BANK CONFLICTS
  Number of actual bank conflicts.
- PE0PE1_11
  0BH MISALIGNED DATA MEMORY OR I/O REFERENCES
  Number of memory or I/O reads or writes that are misaligned.
- PE0PE1_12
  0CH CODE READ
  Number of instruction reads.
- PE0PE1_13
  0DH CODE TLB MISS
  Number of instruction reads that miss the code TLB.

- PE0PE1_14
  0EH CODE CACHE MISS
  Number of instruction reads that miss the internal code cache.

- PE0PE1_15
  0FH ANY SEGMENT REGISTER LOADED
  Number of writes into any segment register in real or protected mode.

- PE0PE1_16
  12H Branches
  Number of taken or not taken branches, including conditional branches, jumps, calls, returns, software interrupts, and interrupt returns.

- PE0PE1_17
  13H BTB_HITS
  Number of BTB hits that occur.

- PE0PE1_18
  14H TAKEN_BRANCH_OR_BTB_HIT
  Number of taken branches or BTB hits that occur.

- PE0PE1_19
  15H PIPELINE FLUSHES
  Number of pipeline flushes that occur.

- PE0PE1_20
  16H INSTRUCTIONS_EXECUTED
  Number of instructions executed (up to two per clock).

- PE0PE1_21
  17H INSTRUCTIONS_EXECUTED_VPIPE
  Number of instructions executed in the V_pipe. It indicated the number of instructions that were paired.

- PE0PE1_22
  18H BUS_CYCLE_DURATION
  Number of clocks while a bus cycle is in progress. This event measures bus use.

- PE0PE1_23
  19H WRITE_BUFFER_FULL_STALL_DURATION
  Number of clocks while the pipeline is stalled due to full write buffers.

- PE0PE1_24
  1AH WAITING_FOR_DATA_MEMORY_READ_STALL_DURATION
  Number of clocks while the pipeline is stalled while waiting for data memory reads.

- PE0PE1_25
  1BH STALL_ON_WRITE_TO_AN_E-_OR_M-STATE_LINE
  Number of stalls on writes to E- or M-state lines..

- PE0PE1_26
  1CH LOCKED_BUS_CYCLE
  Number of locked bus cycles that occur as the result of the LOCK prefix or LOCK instruction, page-table updates, and descriptor table updates.

- PE0PE1_27
  1DH I/O_READ_OR_WRITE_CYCLE
  Number of bus cycles directed to I/O space.

- PE0PE1_28
  1EH NONCACHEABLE_MEMORY_READS
  Number of non-cacheable instruction or data memory read bus cycles.

- PE0PE1_29
  1FH PIPELINE_AGI_STALLS
  Number of address generation interlock (AGI) stalls.

- PE0PE1_30
  22H FLOPS
  Number of floating-point operations that occur. Transcendental instructions consist of multiple adds and multiplies and will signal this event multiple times. Instructions generating the divide-by-zero, negative square root, special operand, or stack exceptions will not be counted. Instructions generating all other floating-point exceptions will be counted. The integer multiply instructions and other instructions which use the FPU will be counted.

- PE0PE1_31
  23H BREAKPOINT_MATCH_ON_DR0_REGISTER
  Number of matches on register DR0 breakpoint.

- PE0PE1_32
  24H BREAKPOINT_MATCH_ON_DR1_REGISTER
  Number of matches on register DR1 breakpoint.

- PE0PE1_33
  25H BREAKPOINT_MATCH_ON_DR2_REGISTER
  Number of matches on register DR2 breakpoint.

- PE0PE1_34
  26H BREAKPOINT_MATCH_ON_DR3_REGISTER
  Number of matches on register DR3 breakpoint.

- PE0PE1_35
  27H HARDWARE_INTERRUPTS
  Number of taken INTR and NMI interrupts.

- PE0PE1_36
  28H DATA_READ_OR_WRITE
  Number of memory data reads and/or writes.

- PE0PE1_37
  29H DATA_READ_MISS_OR_WRITE_MISS
  Number of memory read and/or write accesses that miss the internal data cache.

- **Counter-specific events:**

  - **Specific to counter 0:**

    * PE0_0
      2AH BUS_OWNERSHIP_LATENCY
      The time from LRM bus ownership request to bus ownership granted (*MMX extension*).
    * PE0_1
      2CH CACHE_M-STATE_LINE_SHARING
      Number of times a processor identified a hit to a modified line due to a memory access in the other processor (*MMX extension*).
    * PE0_2
      2DH EMMS_INSTRUCTIONS_EXECUTED
      Number of EMMS instructions executed (*MMX extension*).
    * PE0_3
      2EH BUS_UTILIZATION_DUE_TO_PROCESSOR_ACTIVITY
      Number of clocks the bus is busy due to the processor's own activity (*MMX extension*).
    * PE0_4
      30H NUMBER_OF_CYCLES_NOT_IN_HALT_STATE
      Number of cycles the processor is not idle due to HLT instruction (*MMX extension*).
    * PE0_5
      32H FLOATING_POINT_STALLS_DURATION
      Number of clocks while pipe is stalled due to a floating-point freeze (*MMX extension*).
    * PE0_6
      33H D1_STARVATION_AND_FIFO_IS_EMPTY
      Number of times D1 stage cannot issue ANY instructions since the FIFO buffer is empty (*MMX extension*).

* PE0_7
  35H PIPELINE_FLUSHES_DUE_TO_WRONG_BRANCH_PREDICTIONS
  Number of pipeline flushes due to wrong branch predictions resolved in either the E-stage or the WB-stage (*MMX extension*).
* PE0_8
  37H MISPREDICTED_OR_UNPREDICTED_RETURNS
  Number of returns predicted incorrectly or not predicted at all (*MMX extension*).
* PE0_9
  39H RETURNS
  Number of returns executed (*MMX extension*).
* PE0_10
  3AH BTB_FALSE_ENTRIES
  Number of false entries in the Branch Target Buffer (*MMX extension*).

– **Specific to counter 1:**

* PE1_0
  2AH BUS_OWNERSHIP_TRANSFERS
  Number of bus ownership transfers (*MMX extension*).
* PE1_1
  2CH CACHE_LINE_SHARING
  Number of shared data lines in the L1 cache (*MMX extension*).
* PE1_2
  2EH WRITES_TO_NONCACHEABLE_MEMORY
  Number of write accesses to non-cacheable memory (*MMX extension*).
* PE1_3
  30H DATA_CACHE_TLB_MISS_STALL_DURATION
  Number of clocks the pipeline is stalled due to a data cache translation look-aside buffer miss (*MMX extension*).
* PE1_4
  31H TAKEN_BRANCHES
  Number of branches taken (*MMX extension*).
* PE1_5
  33H D1_STARVATION_AND_ONLY_ONE_INSTRUCTION_IN_FIFO
  Number of times the D1 stage issues just a single instruction since the FIFO buffer had just one instruction ready (*MMX extension*).
* PE1_6
  35H PIPELINE_FLUSHES_DUE_TO_WRONG_BRANCH_PREDICTIONS_RESOLVED_IN_WB-STAGE
  Number of pipeline flushes due to wrong branch predictions resolved in the WB-stage (*MMX extension*).
* PE1_7
  37H PREDICTED_RETURNS
  Number of predicted returns (*MMX extension*).
* PE1_8
  3AH BTB_MISS_PREDICTION_ON_NOT_TAKEN_BRANCH
  Number of times the BTB predicted a not-taken branch as taken (*MMX extension*).

By default, the instructions *RDMSR* and *WRMSR* to access the performance counter registers are kernel-mode instructions (ring 0).

In [12] are software tools concerning the performance counters on Pentium-like processors described. On Linux systems, libpperf is available to access the performance counters. It was written by M. Patrick Goda and Michael S. Warren from Los Alamos National Laboratory. *libpperf* itself is based on the msr device implemented by Stephan Meyer for Linux 2.0.x and 2.1.x.

A.5.2 Intel PentiumPro/Pentium II/Pentium III

To keep binary compatibility with the predecessor processors, the PentiumPro, Pentium II, and Pentium III have 8 registers, 32 bit width each. First level cache is 8 KB for instructions (ICache) and 8 KB for data (DCache) on PentiumPro, and 16 KB for both caches on Pentium II and Pentium III. As the PentiumPro, Pentium II, and Pentium III are CISC-microprocessors (complex instruction set computer), every instruction

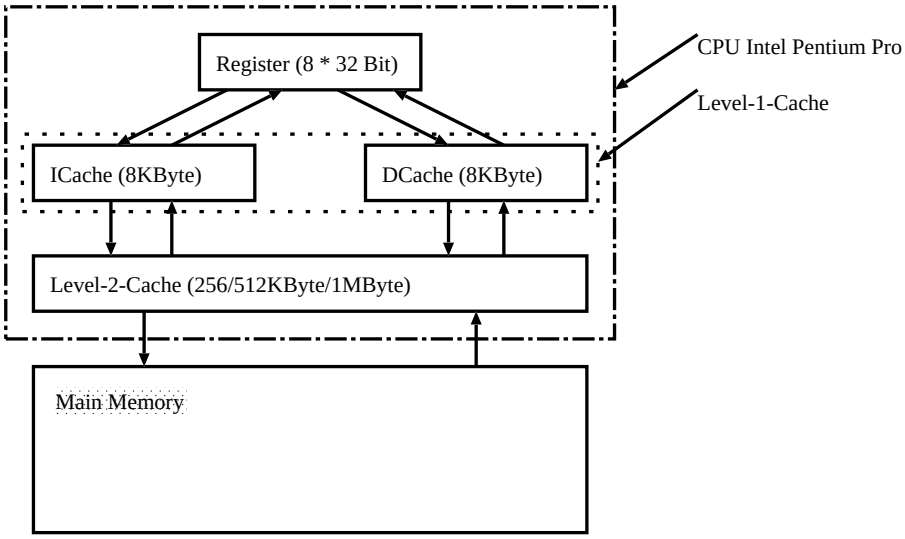

Figure A.5: Memory hierarchy of the Intel PentiumPro

is divided internally into micro-operations (UOP's) of fixed length. Dependent on the complexity of the instruction, the instruction is divided into 1-4 UOP's.

The PentiumPro, Pentium II, and Pentium III has 2 performance counters capable of counting a total of 77 different events (at most two at a time), some of them with an additional unit mask as parameter to further subdivide the event type. Some of the events are countable only by a specific counter. The Pentium III has 4 additional events concerning Streaming SIMD Extensions. The events countable by both counters are:

- PP0PP1_0
  43H DATA_MEM_REFS
  All memory references, both cacheable and non-cacheable.

- PP0PP1_1
  45H DCU_LINES_IN
  Number of allocated lines in the 1st level data cache.

- PP0PP1_2
  46H DCU_M_LINES_IN
  Number of allocated lines in the 1st level data cache which have the status *modified*.

- PP0PP1_3
  47H DCU_M_LINES_OUT
  Number of evicted lines in the 1st level data cache which were marked as *modified*.

- PP0PP1_4
  48H DCU_MISS_OUTSTANDING
  Weighted number of cycles while a 1st level data cache miss is outstanding. An access that also misses the L2 is short-changed by 2 cycles. (i.e. if counts N cycles, should be N+2 cycles.) Subsequent loads to the same cache line will not result in any additional counts. Count value not precise, but still useful.

- PP0PP1_5
  80H IFU_IFETCH
  Number of 1st level instruction cache loads.

- PP0PP1_6
  81H IFU_IFETCH_MISS
  Number of 1st level instruction cache misses.

- PP0PP1_7
  85H ITLB_MISS
  Number of instruction transfer look-aside buffer misses.

- PP0PP1_8
  86H IFU_MEM_STALL
  Number of cycles in which the instruction fetch pipe stage is stalled.

- PP0PP1_9
  87H ILD_STALL
  Number of cycles the instruction length decoder is stalled.

- PP0PP1_10
  28H L2_IFETCH
  Number of instruction fetches from the 2nd level cache.

- PP0PP1_11
  29H L2_LD
  Number of data loads from the 2nd level cache.

- PP0PP1_12
  2AH L2_ST
  Number of data stores to the 2nd level cache.

- PP0PP1_13
  24H L2_LINES_IN
  Number of lines allocated in the 2nd level cache.

- PP0PP1_14
  26H L2_LINES_OUT
  Number of cache lines removed from the 2nd level cache.

- PP0PP1_15
  25H L2_M_LINES_INM
  Number of allocated cache lines in the 2nd level cache which have been modified.

- PP0PP1_16
  27H L2_M_LINES_OUTM
  Number of modified cache lines in the 2nd level cache which have been removed.

- PP0PP1_17
  2EH L2_RQSTS Number of requests to the 2nd level cache.

- PP0PP1_18
  21H L2_ADS
  Number of address strobes at 2nd level cache address bus.

- PP0PP1_19
  22H L2_DBUS_BUSY
  Number of cycles during which the data bus was busy.

- PP0PP1_20
  23H L2_DBUS_BUSY_RD
  Number of cycles during which the data bus was busy transferring data from 2nd level cache to the processor.

- PP0PP1_21
  62H BUS_DRDY_CLOCKS
  Number of cycles the DRDY-signal was active.

- PP0PP1_22
  63H BUS_LOCK_CLOCKS
  Number of processor clock cycles during which the LOCK-signal is asserted.

- PP0PP1_23
  60H BUS_REQ_OUTSTANDING
  Number of outstanding bus requests which either result out from a cacheable read request of 1st level data cache lines or a to be completed bus operation.

- PP0PP1_24
  65H BUS_TRAN_BRD
  Number of burst read transactions.

- PP0PP1_25
  66H BUS_TRAN_RFO
  Number of read for ownership transactions.

- PP0PP1_26
  67H BUS_TRANS_WB
  Number of write back transactions.

- PP0PP1_27
  68H BUS_TRAN_IFETCH
  Number of completed instruction fetch transactions.

- PP0PP1_28
  69H BUS_TRAN_INVAL
  Number of completed bus invalidate transactions.

- PP0PP1_29
  6AH BUS_TRAN_PWR
  Number of completed partial write transactions.

- PP0PP1_30
  6BH BUS_TRANS_P
  Number of completed partial transactions.

- PP0PP1_31
  6CH BUS_TRANS_IO
  Number of completed I/O transactions.

- PP0PP1_32
  6DH BUS_TRAN_DEF
  Number of completed deferred transactions.

- PP0PP1_33
  6EH BUS_TRAN_BURST
  Number of completed burst transactions.

- PP0PP1_34
  70H BUS_TRAN_ANY
  Number of all completed transactions.

- PP0PP1_35
  6FH BUS_TRAN_MEM
  Number of completed memory transactions.

- PP0PP1_36
  64H BUS_DATA_RCV
  Number of bus clock cycles during which this processor is receiving data.

- PP0PP1_37
  61H BUS_BNR_DRV
  Number of bus clock cycles during which this processor is driving the BNR pin.

- PP0PP1_38
  7AH BUS_HIT_DRV
  Number of bus clock cycles during which this processor is driving the HIT pin including cycles due to snoop stalls.

- PP0PP1_39
  7BH BUS_HITM_DRV
  Number of bus clock cycles during which this processor is driving the HITM pin including cycles due to snoop stalls.

- PP0PP1_40
  7EH BUS_SNOOP_STALL
  Number of clock cycles during which the bus is snoop stalled.

- PP0PP1_41
  03H LD_BLOCKS
  Number of store buffer locks.

- PP0PP1_42
  04H SB_DRAINS
  Number of cycles in which the store buffer blocks.

- PP0PP1_43
  05H MISALIGN_MEM_REF
  Number of misaligned data memory references.

- PP0PP1_44
  C0H INST_RETIRED
  Number of instructions retired.

- PP0PP1_45
  C2H UOPS_RETIRED
  Number of micro-operations retired.

- PP0PP1_46
  D0H INST_DECODER
  Number of instructions decoded and translated to UOP's.

- PP0PP1_47
  C8H HW_INT_RX
  Number of hardware interrupts received.

- PP0PP1_48
  C6H CYCLES_INT_MASKED
  Number of processor cycles for which interrupts are disabled.

- PP0PP1_49
  C7H CYCLES_INT_PENDIND_AND_MASKED
  Number of ptrocessor cycles for which interrupts are disabled and interrupts are pending.

- PP0PP1_50
  C4H BR_INST_RETIRED
  Number of branch instructions retired.

- PP0PP1_51
  C5H BR_MISS_PRED_RETIRED
  Number of completed but mispredicted branches.

- PP0PP1_52
  C9H BR_TAKEN_RETIRED
  Number of completed taken branches.

- PP0PP1_53
  CAH BR_MISS_PRED_TAKEN_RET
  Number of completed taken, but mispredicted branches.

- PP0PP1_54
  E0H BR_INST_DECODED
  Number of decoded branch instructions.

- PP0PP1_55
  E2H BTB_MISSES
  Number of branches that missed the BTB.

- PP0PP1_56
  E4H BR_BOGUS
  Number of bogus branches.

- PP0PP1_57
  E6H BACLEARS
  Number of times BACLEAR-signal is asserted.

- PP0PP1_58
  A2H RESOURCE_STALLS
  Number of cycles during which there are resource related stalls.

- PP0PP1_59
  D2H PARTIAL_RAT_STALLS
  Number of cycles or events for partial stalls.

- PP0PP1_60
  06H SEGMENT_REG_LOADS
  Number of segment register loads.

- PP0PP1_61
  79H CPU_CLK_UNHALTED
  Number of cycles during which the processor is not halted.

- PP0PP1_62
  B0H MMX_INSTR_EXEC
  Number of MMX-instructions executed.

- PP0PP1_63
  B3H MMX_INSTR_TYPE_EXEC
  Number of MMX-instructions executed. The further parameter unit mask specifies which category should be counted.

- PP0PP1_64
  B1H MMX_SAT_INSTR_EXEC
  MMX saturated instructions executed.

- PP0PP1_65
  B2H MMX_uOPS_EXEC
  Number of MMX uops executed.

- PP0PP1_66
  CCH FP_MMX_TRANS
  Transitions from MMX instructions to FP instructions.

- PP0PP1_67
  CDH MMX_ASSIST
  Number of MMX assists (EMMS instructions executed).

- PP0PP1_68
  CEH MMX_INSTR_RET
  Number of MMX instructions retired.

- PP0PP1_69
  D4H SEG_RENAME_STALLS
  Segment register renaming stalls.

- PP0PP1_70
  D5H SEG_REG_RENAMES
  Segment registers renamed.

- PP0PP1_71
  D6H RET_SEG_RENAMES
  Number of segement register rename events retired.

- PP0PP1_72
  D8H EMON_SSE_INST_RETIRED
  Number of Streaming SIMD extensions retired.

- PP0PP1_73
  D9H EMON_SSE_COMP_INST_RET
  Number of Streaming SIMD Extensions computation instructions retired.

- PP0PP1_74
  07H EMON_SSE_PRE_DISPATCHED
  Number of prefetch/weakly ordered instructios dispatched (inclusive speculative prefetches).

- PP0PP1_75
  4BH EMON_SSE_PRE_MISS
  Number of prefetch/weakly-ordered instructions that miss all caches.

- **Counter-specific events:**

  - **Specific to counter 0:**

    * PP0_0
      C1H FLOPS
      Number of retired floating point instructions.
    * PP0_1
      10H FP_COMP_OPS_EXE
      Number of floating point operations started (but which may not have been all completed.)
    * PP0_2
      14H CYCLES_DIV_BUSY
      Number of cycles during which the divider is busy.

  - **Specific to counter 1:**

    * PP1_0
      11H FP_ASSIST
      Number of floating-point exception cases handled by microcode.
    * PP1_1
      12H MUL
      Number of multiplies (integer and floating-point).
    * PP1_2
      13H DIV
      Number of divides (integer and floating-point).

All of the events can be counted on PentiumPro as well as on Pentium II and Pentium III. The Pentium II and Pentium III have additional events defined mainly for MMX-extensions [13].

The same remarks as stated above in the Pentium-section concerning software environments apply to the Pentium Pro, Pentium II, and Pentium III as well.